Paavo CAMPS
Vaibhav ARORA

Query Optimization (Ex-3)
PRE-3 DBMS

2020-11-04

1 Introduction

The following work summarizes the various methods used for optimizing the queries in the project for the IMDB Data-set (see Figure 1). In particular, we use alternative equivalent queries, create indexes when suitable (clustered and non-clustered) and materialized view.

It is known that running a query for the first time, the server reads all the data from disk and thus is time-consuming (cold-run). For consecutive runs of the same query, the server can refer to part-cached data and so the execution time is faster (hot-run).
Therefore, we base the execution profiling of each query on a cold-run and an average over 3 hot runs. We then base our optimization results on the hot run. We feel this is a representative way because queries written for a database like IMDB are unlikely to have a fully cold cache, and hot caches are more representative of its use.

Now, we know there are two types of cache: shared buffers postgres cache and OS cache. In PostgreSQL, we do not have any predefined functionality to clear the cache from the memory. To clear the database level cache, we need to:
- shut down the whole instance OR
- use DISCARD TEMP (drops all temporary tables created in the current session) and DISCARD PLAN (releases all cached query plans)

To clear the operating system cache, since we're using a Windows machine, unlike Linux, there is no one complete way of clearing OS cache from applications, and thus we need to:
- use the operating system utility commands, settings (smart storage), and disk cleaner
- restart the machine (the reliable way to control the experiment)

We therefore define our experiment procedure as follows:
Step 1: Restart the machine
Step 2: Run a query in psql and store its results as part of cold-run
Step 3: Run the same query 3 more times and store the average of the results as part of hot-run
Step 4: Analyze the query plan to look for improvement
Step 5: Repeat Step 1-4 for other queries (base and optimized ones)

NB:
1. By 'store the results' we mean generating a text file of the query plan and using https://explain.dalibo.com to generate the graphical query plan and create a shareable link for the same.
2. We consider the 'execution time' of the query as the metric for performance.
3. Since AutoVacuum was on by default, therefore the table statistics are already stored available for the query optimizer for all queries and tables.
4. No other activities are performed and the charger is connected whilst running the queries.

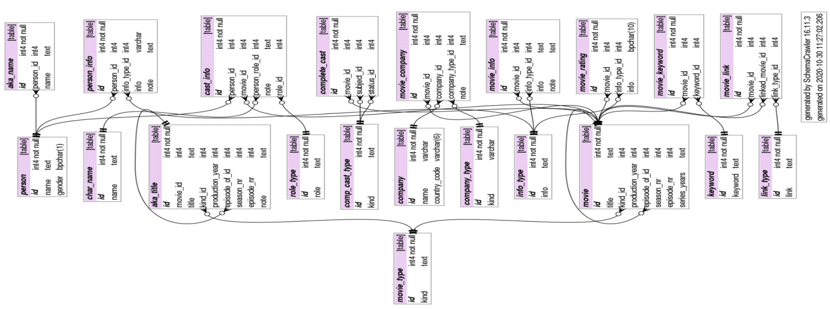


Figure 1: Schema of the IMDB dataset

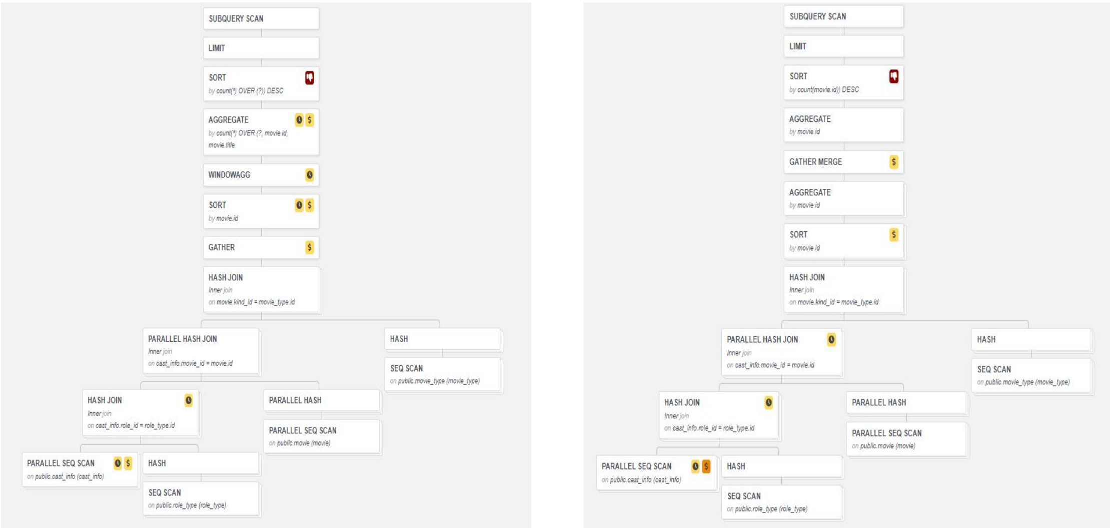
2 System Specifications

System:
Processor: Intel Core i7-8550U CPU @ 1.80GHz 1.99 GHz
Installed RAM: 8.00 GB
System type: 64-bit operating system
Edition: Windows 10 Home
Disk Sequential 64.0 Read: 128.05 MB/s Disk Random 16.0 Read: 1.49 MB/s

PostgreSQL:
version 13.0

3 Query Optimization

3.1 Equivalent queries: Query-6



(a) Using Windows Function (b) Using GroupBy

Figure 2: Query Plans for query 6

Node Type	Count	Time	
Aggregate	1	2s 461ms	21%
Hash Join	3	2s 344ms	20%
Seq Scan	4	2s 164ms	18%
Sort	2	1s 624ms	14%
WindowAgg	1	1s 388ms	12%
Gather	1	1s 138ms	10%
Hash	3	437ms	4%
Subquery Scan	1	0.004ms	0%
Limit	1	0.002ms	0%

(a) Using Windows Function

Node Type	Count	Time	
Hash Join	3	2s 188ms	37%
Seq Scan	4	1s 858ms	31%
Aggregate	2	646ms	11%
Sort	2	585ms	10%
Gather Merge	1	390ms	7%
Hash	3	237ms	4%
Subquery Scan	1	0.006ms	0%
Limit	1	0.004ms	0%

(b) Using GroupBy

Figure 3: Per Node Stats for query 6 during a particular hot run

Query-6 was originally written using Windows Function. The Aggregation in the query with the Windows Function is slower (see Figure 3) since it expands each row into the set of rows that represent the window associated with it, and then performs the aggregation. This leads to more reads and writes to the tempdb before being sorted and is thus by nature, slow.

It is possible to eliminate multiple read-writes. This can be done by an equivalent query using GroupBy can be written wherin the I/O operations happen once for Aggregation. This leads to better performance as can be seen from the results of the average run-time of hot-runs.

Table 1

	Cold	Hot-1	Hot-2	Hot-3	Hot-Avg
Over	120s	12s	10.8s	10.6s	11.13s
GroupBy	119s	5.9s	7.7s	5.8s	6,46s

3.2 Indexes: Query-1

For Query-1, as can be seen from Figure 4, due to the nature of the query (WHERE clause filters specific rows on a very large table), the sequential scans to find the relevant records (person.name = ’Wayne, John’ or person.name = ’Streep, Meryl’) and similarly to form the joined table, an immediate performance improvement can be expected using indexes.

Therefore, in accordance to the WHERE clause of the query, we clustered person.name for the person table, and indexed on personInfo(personId, infoId). The improvement in query execution time is almost a 1000-fold as can be seen from the results in Table 2.

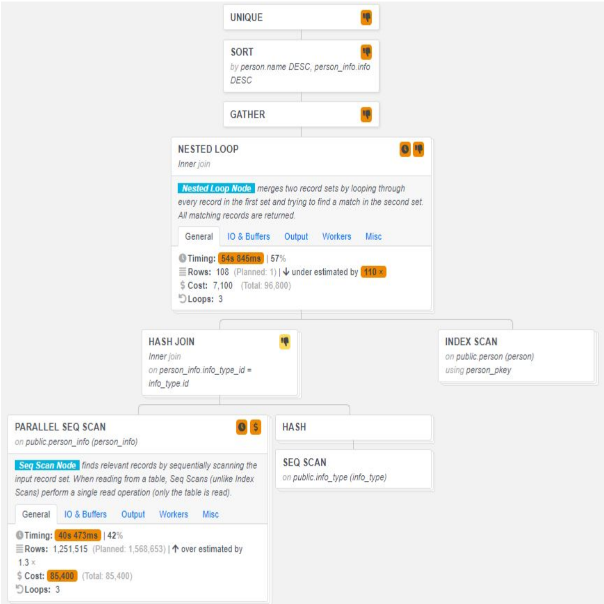


Figure 4: Query plan for query-1

Table 2

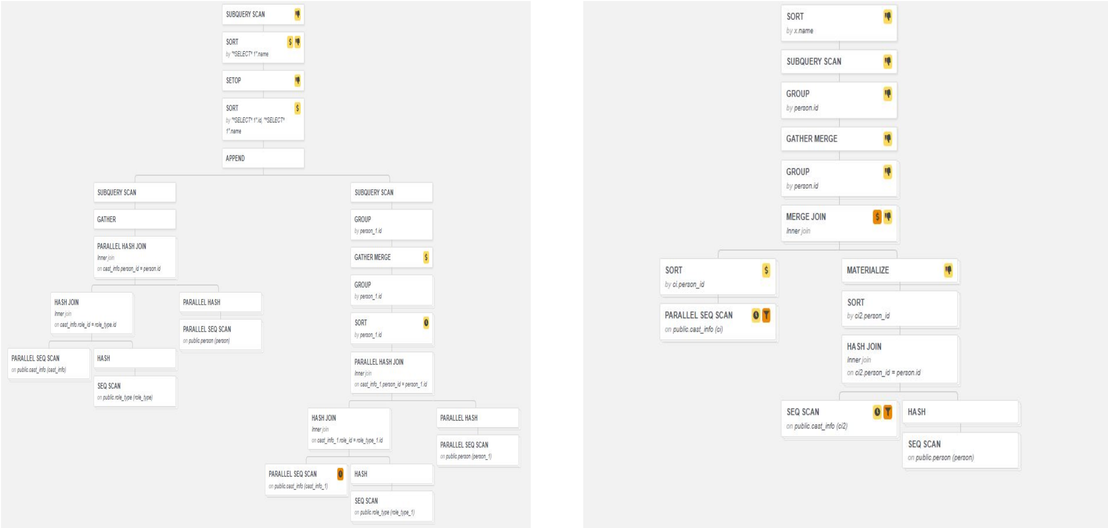
	Cold	Hot-1	Hot-2	Hot-3	Hot-Avg
Normal	96s	1128ms	1130ms	1140ms	1132ms
Index	11ms	2.4ms	1.4ms	1.5ms	1.8ms

3.3 Materialized View - Query 4

Query-4 was originally written using INTERSECTION, which after analyzing the query plan, was found to have room for improvement (reads happening in 2 parts). The query was improved by using ALIASES which reduced the execution time by almost half. Since the nature of the query was to scan through thousands of rows but return a handful, creating a materialized view of the subquery used in Query-4 brought down the execution speed from 71.6s to 13.6ms (see Table 3).

Table 2

	Cold	Hot-1	Hot-2	Hot-3	Hot-Avg
Intersection	3037s	210s	87s	59s	118.6s
Alias	170s	136s	43s	36s	71.6s
Material View	17.4	13.4ms	13.5ms	14ms	13.6ms



(a) Using Intersect (b) Using Alias

Figure 5: Query plans for query 4 during a particular hot run

4 Comparison to Pandas

We created a jupyter notebook in which we imported the whole database into pandas dataframes. Then we recreated Query-1 in python with pandas and measured the time.
Load-time of all tables = 342.194s
Load-time of required tables = 13.837s
Execution-time = 0.787s

Next we used pandas.readsqlquery after initializing the connection to postgreSQL to run Query-1.
Execution-time = 0.0518s

Thus we see that atleast for this particular query postgreSQL is almost 15x faster. It can be safely concluded that postgreSQL is way faster to use than pandas to achieve the same goal.

5 APPENDIX

Equivalent queries - Query 6
Original query with Over()
Cold-run: https://explain.dalibo.com/plan/Uptv
Hot-run1: https://explain.dalibo.com/plan/c0
Hot-run2: https://explain.dalibo.com/plan/KVs
Hot-run3: https://explain.dalibo.com/plan/PRK

Equivalent query using GroupBy
Cold-run: https://explain.dalibo.com/plan/49c
Hot-run 1: https://explain.dalibo.com/plan/LCq
Hot-run 2: https://explain.dalibo.com/plan/axt
Hot-run 3: https://explain.dalibo.com/plan/XhD

Indexes - Query 1
Original query
Cold-run: https://explain.dalibo.com/plan/K8V
Hot-run 1: https://explain.dalibo.com/plan/gtT
Hot-run 2: https://explain.dalibo.com/plan/pX0
Hot-run 3: https://explain.dalibo.com/plan/3YW

Query with clustered person table on name with ordered index and indexed on personInfo(personId, infoId)
Cold-run: https://explain.dalibo.com/plan/xDD
Hot-run 1: https://explain.dalibo.com/plan/cua
Hot-run 2: https://explain.dalibo.com/plan/1BP
Hot-run 3: https://explain.dalibo.com/plan/sTC

Materialized View - Query 6
Original query using Intersection
Cold-run: https://explain.dalibo.com/plan/ddb
Hot-run1: https://explain.dalibo.com/plan/v1X
Hot-run2: https://explain.dalibo.com/plan/s93
Hot-run3: https://explain.dalibo.com/plan/AJk

Equivalent query using aliases
Cold-run: https://explain.dalibo.com/plan/eEB
Hot-run1: https://explain.dalibo.com/plan/6nu
Hot-run2: https://explain.dalibo.com/plan/hX8
Hot-run3: https://explain.dalibo.com/plan/mua

Query using Materialized View
Cold-run: https://explain.dalibo.com/plan/gpA
Hot-run1: https://explain.dalibo.com/plan/pN2
Hot-run2: https://explain.dalibo.com/plan/k52
Hot-run3: https://explain.dalibo.com/plan/OD2n